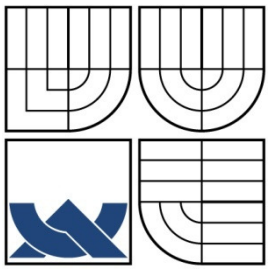
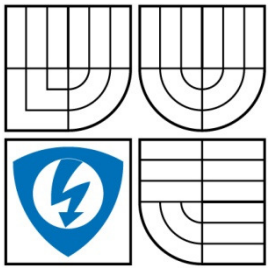


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

# KÓDOVÁNÍ PRO PŘENOSOVÉ SYSTÉMY SE ZPĚTNÝM KANÁLEM

CODING FOR TRANSMISSION SYSTEMS WITH FEEDBACK CHANNEL

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JAN VARMUŽA
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. KAREL NĚMEC, CSc
SUPERVISOR

BRNO 2008

# LICENČNÍ SMLOUVA
# POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

**1. Pan/paní**

Jméno a příjmení: Jan Varmuža

Bytem: Lípová 65, 66449, Ostopovice

Narozen/a (datum a místo): 15. 8. 1984 v Brně

(dále jen „autor“)

a

**2. Vysoké učení technické v Brně**

Fakulta elektrotechniky a komunikačních technologií

se sídlem Údolní 244/53, 602 00, Brno

jejímž jménem jedná na základě písemného pověření děkanem fakulty:

prof. Ing. Kamil Vrba, CSc.

(dále jen „nabyvatel“)

## Čl. 1
## Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - □ disertační práce
   - □ diplomová práce
   - ■ bakalářská práce
   - □ jiná práce, jejíž druh je specifikován jako .....

   (dále jen VŠKP nebo dílo)

Název VŠKP: Kódování pro přenosové systémy se zpětným kanálem

Vedoucí/ školitel VŠKP: doc. Ing. Karel Němec, CSc.

Ústav: Ústav telekomunikací

Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v*:

- ■ tištěné formě – počet exemplářů 1
- ■ elektronické formě – počet exemplářů 1

* hodící se zaškrtněte

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

**Článek 2**
**Udělení licenčního oprávnění**

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - ▪ ihned po uzavření této smlouvy
   - □ 1 rok po uzavření této smlouvy
   - □ 3 roky po uzavření této smlouvy
   - □ 5 let po uzavření této smlouvy
   - □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

**Článek 3**
**Závěrečná ustanovení**

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: …….

…….. ……
Nabyvatel Autor

**POPISNÝ SOUBOR ZÁVĚREČNÉ PRÁCE**

| | |
|---|---|
| Autor: | Jan Varmuža |
| Název závěrečné práce: | Kódování pro přenosové systémy se zpětným kanálem |
| Typ závěrečné práce: | bakalářská práce |
| Vedoucí závěrečné práce: | doc. Ing. Karel Němec, CSc. |
| Škola: | Vysoké učení technické v Brně |
| Fakulta: | Fakulta elektrotechniky a komunikačních technologií |
| Ústav / ateliér: | Ústav telekomunikací |
| Studijní program: | Elektrotechnika, elektronika, komunikační a řídicí technika |
| Studijní obor: | Teleinformatika |

**Anotace závěrečné práce:**
Práce se zabývá výběrem a implementací vhodného kódu, který je schopen zabezpečit digitální přenos proti shlukujícím se chybám. Po krátkém teoretickém úvodu následuje výběr vhodného kódu, návrh jeho použití a experimentální ověření funkčnosti zvoleného systému pomocí počítačové simulace.

**Klíčová slova:** kódování, kodér, dekodér, kodek, přenos, shluk chyb.

**SUMMARY**

| | |
|---|---|
| Author: | Jan Varmuža |
| Title: | Coding for Transmission Systems with Feedback Channel |
| Type of final project: | Bachelor´s Thesis |
| Supervisor: | doc. Ing. Karel Němec, CSc. |
| University: | Brno University of Technology |
| Faculty: | Faculty of Electrical Engineering and Communication |
| Department: | Department of Telecommunications |
| Programme title: | Electrical, Electronic, Communication and Control Technology |
| Branch: | Teleinformatics |

**Abstract:**
This project deals with selection and implementation of coding system, which can secure digital transmission from bursts of errors. Following short theoretical introduction is selection of proper coding. A next chapter deals with design and experimental system functionality verification using computer simulation.

**Keywords:** coding, coder, decoder, codec, transmission, burst of errors.

**NÁZEV TÉMATU:**

# KÓDOVÁNÍ PRO PŘENOSOVÉ SYSTÉMY SE ZPĚTNÝM KANÁLEM

**POKYNY PRO VYPRACOVÁNÍ:**

Vypracujte návrh protichybového kodeku, který zabezpečí digitální přenos dat proti shlukujícím se chybám pomocí detekčního kódu s opravou opakováním chybného přenosu. Distribuce chyb v kanálu zadána takto: Bitová chybovost $p_b$ = 2,1 .10-6 chyb/bit, délka shluku chyb b je menší nebo rovna 22 bitu, délka bezchybného intervalu A je větší nebo rovna 1800 bitu. Ověřte funkční schopnosti tohoto protichybového kodeku metodou, kterou považujete pro tento návrh za nejvhodnější.

**DOPORUCENÁ LITERATURA:**

[1] OOI,J.M. Coding for channels with feedback. Kluwer Academic Publisher. Boston, Dortrecht, London, 1999, ISBN 0-7923-8207-2.

[2] HOUGHTON,A. Error Coding for Engineers. Kluwer academic Publishers. Boston, Dordrecht, London. 2001.

**Prohlášení**

Prohlašuji, že svou bakalářskou práci na téma *Kódování pro přenosové systémy se zpětným kanálem* jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne .....

podpis autora

**Poděkování**

Děkuji vedoucímu bakalářské práce doc. Ing. Karlu Němcovi, CSc. za odbornou pomoc a rady při zpracování mé bakalářské práce.

# Obsah

# 1 Teoretický úvod a popis používaných typů kódování

## 1.1 Úvod

Cílem této práce je vypracovat obecný popis systému, který je schopen zabezpečit digitální přenos dat proti shlukujícím se chybám. Zabezpečení má být provedeno pomocí detekčního kódu, chybný přenos má být opakován. Jsou zadány parametry kanálu a chybovost, kterou musí být systém schopen zabezpečit. Závěrem bude provedena simulace, ve které ověříme funkční vlastnosti navrženého systému.

## 1.2 Prvotní výběr kódu

Na začátku byl proveden krátký rozbor nejznámějších kódů a byla ověřována jejich teoretická schopnost zabezpečit shluk chyb dlouhý 22 bitů.

**Lineární blokové kódy**

Při tvorbě vytvářecí matice se vychází ze znalosti vztahu mezi zabezpečovacími vlastnostmi kódu a Hammingovou vzdáleností $d_{min}$. Minimální délka kódového slova a počet zabezpečovacích prvků určíme pomocí Plotkinova vztahu.

$$n \geq \frac{F.d_{min} - 1}{F - 1} = \frac{F.(2t + 1) - 1}{F - 1} = \frac{2\,.\,(2\,.\,10 + 1) - 1}{2 - 1} = 89\ bit\text{ů}$$

Stanovme si tedy n = 90. Dále si dle následujícího vztahu vypočteme počet zabezpečovacích prvků.

$$r \geq n - k = 90 - (1 + log_2 45) \approx 84\ bit\text{ů}$$

Takovýto kód je však naprosto nepoužitelný, neboť zabezpečovat 5 informačních bitů 84 bity redundantními je velmi neekonomické. Bylo by sice možné zvětšit délku kódové kombinace ale přesto je 84 zabezpečujících bitů příliš hodně.

Závěr: Kód má příliš velkou redundanci, nelze použít.

**Paritní kód**

Nejjednodušší detekční systematický kód, detekuje pouze jednonásobnou chybu.

Závěr: Kód nelze použít.

**Hammingovy blokové kódy**

Typické blokové korekční kódy, $d_{min}$=3 nebo 4, lze je použít k detekci až dvou chyb.

Závěr: Kód nelze použít, příliš malá detekční schopnost.

**Cyklický detekční kód podle doporučení V.41**

Je určen vytvářecí maticí $G(x) = x^{16}+x^{12}+x^5+1$. Detekuje každou lichou chybu nebo shluk chyb, jehož délka nepřekročí 16 bitů.

Závěr: Kód má nedostatečnou detekční schopnost, nelze tedy použít. Je však možno použít jiný cyklický kód schopný detekovat delší shluk chyb.

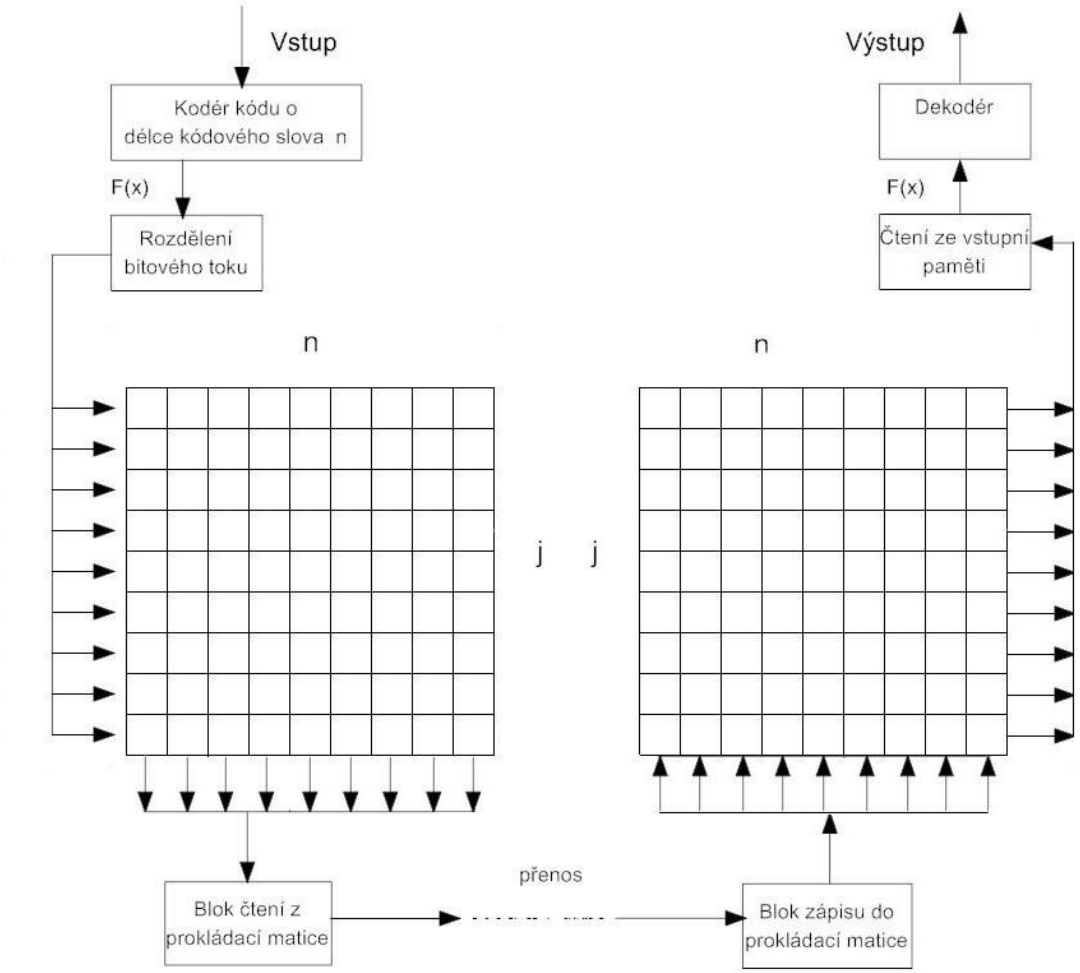
**Blokové prokládání**

Byla otestována možnost použití blokového prokládání. Přestože toto řešení nepatří ke standardním, je možno přenos takto zabezpečit.

Závěr: Není příliš vhodné kvůli zpoždění vzniklé použitím maticových prokládacích paměti. Tato varianta však byla dále zkoumána, především jako referenční k vybranému typu cyklických kódů.

## 1.3 Blokové prokládání

Obecně se jedná o systémy s prokládáním. Někdy dochází ke shlukům chyb takové délky, že nemáme k dispozici kód, který by byl schopen zabezpečit bezchybný přenos v daném systému. Využíváme proto princip rozprostření informace v čase. Jedním z postupů je změna polohy signálových prvků nezabezpečeného signálového toku tak, aby se změnil charakter rozložení chyb z chyb shlukujících na chyby nezávislé. Tato metoda však způsobuje jisté zpoždění informace při průchodu systémem.




**Obr. 1:** Schematické znázornění principu bitového prokládání

Nezabezpečený tok bitů zabezpečíme jednoduchým např. blokovým kódem, který je schopen detekovat násobnou chybu ve své kódové kombinaci. Tyto zabezpečené kombinace jsou ukládány do maticové paměti o rozměru *n* . *j* tak, že na každém řádku je jedna. Po naplnění matice jsou data odesílána po sloupcích. V případě výskytu shluku chyb dojde k rozložení chyby na vícero kódových slov, v krajním případě na všechna kódová slova odeslaná v rámci jedné sekvence z výstupní maticové paměti. V dekodéru je přijatý datový tok uložen po sloupcích do vstupní maticové paměti stejných rozměrů. Po naplnění matice se po řádcích odesílají přijaté kódové kombinace do dekodéru kódu, který určí bezchybnost přenosu. Pokud byl přenesený datový tok napaden shlukovou chybou, díky použitému systému jsme dosáhli změny chyb na nezávislé. Je sice napadeno *j* kódových kombinací ale s *j*-krát menším počtem chyb.

Tento systém můžeme popsat parametry vlastního kódu a tzv. prokládacím faktorem *j*, který udává počet kódových kombinací, které mají být proloženy. Zpoždění přenosu bitového toku je určeno

kapacitou prokládacích matic na straně kodéru i dekodéru $D_t = 2.j.n.$ Pro stanovení rozměrů maticové paměti musíme znát délku shluku chyb *b* a minimální délku bezchybného intervalu *A*, které zjistíme při statistickém rozboru chyb v kanálu.

Počet řádků *j* matice $j \geq \frac{b}{t}$

*kde t je počet nezávislých chyb opravitelných (detekovatelných) použitým kódem.*

Při stanovení počtu sloupců n hledáme takový kód, který koriguje *t* chyb v kódové kombinaci délky *n* bitů a musí být splněna nerovnost: $n \leq \frac{A+b}{j}$

## 1.4 Cyklické kódy

Základní myšlenka zabezpečení pomocí těchto kódů je velmi prostá - používá zbytků po dělení. Cyklické kódy lze popsat pomocí algebry mnohočlenů, dvojkové číslice vyjádříme jako koeficienty mnohočlenů. Logická jednička je vyjádřena jedničkovým koeficientem u příslušného členu polynomu, nula pak nulovým koeficientem. Zapisujeme zpravidla od nejvyššího členu k nejnižšímu. Matematické operace s mnohočleny se řídí pravidly obyčejné algebry, pouze sčítání se provádí pomocí modulo2.

Cyklický kód je lineární kód definovaný vytvářecím mnohočlenem G(x) stupně *n-k*. Generující polynom nesmí být beze zbytku dělitelný jiným polynomem nižšího řádu, jinak by byl koeficient nultého řádu každého kódového mnohočlenu roven 0. Musí se jednat o takzvaný primitivní polynom. Polynom stupně *m* je primitivní pokud dělí beze zbytku ($x^e+1$) , kde $e=2^m-1$ a nedělí beze zbytku ($x^i+1$), pro všechna $i<k$. Pro každé pozitivní *k*, existuje alespoň jeden primitivní polynom řádu *k*. Důkaz těchto tvrzení je nad rámec možností této práce, je možno ho nalézt v [5] nebo [9].

**Kódování**

Vlastní zakódování provedeme takto. K nezabezpečenému mnohočlenu P(x) připojíme n-k nul a vytvoříme si tak prostor pro připojení zbytku po dělení. V matematické rovině tato operace odpovídá vynásobení polynomu P(x) výrazem ($x^{n-k}$). Tento výraz vydělíme generujícím polynomem G(x), čímž získáme zbytek po dělení R(x), který přičteme k součinu P(x). ($x^{n-k}$). Vyjádřeno rovnicí: $\frac{(x^{n-k}).P(x)}{G(x)} = M(x) + \frac{R(x)}{G(x)}$.

Vynásobíme-li tuto rovnici G(x) dostaneme $F(x) = (x^{n-k}) . P(x) = M(x) . G(x) + R(x)$.

V aritmetice modulo2 je součet roven rozdílu, dále vypustíme člen M(x) . G(x) a rovnici upravíme do tvaru $F(x) = (x^{n-k}) . P(x) + R(x)$.

Mnohočlen zbytku R(x) je stupně n-k a ($x^{n-k}$) . P(x) má nulové koeficienty na posledních *m* místech. Znamená to, že *k* nejvyšších koeficientů F(x) je shodných s koeficienty P(x). Posledních m symbolů v F(x) tvoří vlastní zabezpečovací bity a odpovídají R(x).

**Princip zjišťování chyb**

Přijatá zpráva, která obsahuje chyby, se dá vyjádřit polynomem H(x). Definice přijaté zprávy je H(x) = F(x) + E(x). Jestliže přijatá H(x) není dělitelná generujícím mnohočlenem G(x) beze zbytku, vyskytla se během přenosu chyba. V opačném případě, má li polynom syndromu S(x), což je zbytek po dělení H(x)/G(x), všechny koeficienty nulové, můžeme tvrdit, že přijatá zpráva není poškozena chybou, nebo se jedná o chybu nezjistitelnou.

Pro detekci chyb platí několik následujících tvrzení, které si nyní uvedeme. Tyto podmínky musíme dodržet a později nám pomohou při detekci chyb dle zadání.

1) *Cyklický kód vytvořený jakýmkoliv generujícím polynomem G(x) s více než jedním členem detekuje všechny jednoduché chyby.*

Předpokládejme chybu na pozici *i* zakódované zprávy, kterou vyjádříme chybovým mnohočlenem $x^i$. Žádný mnohočlen s více než jedním členem nedělí $x^i$ beze zbytku. A protože G(x) obsahuje vždy víc než jeden člen, je detekce jednoduché chyby vždy zaručena.

2) *Kód vytvořený generujícím mnohočlenem G(x) detekuje všechny dvojité chyby, pokud délka zakódované zprávy n není větší než exp. e, ke kterému G(x) patří.*

Exponent e je nejmenší celé kladné číslo takové, že G(x) beze zbytku dělí ($x^e$+1). Pro jakékoliv *m* existuje nejméně jeden mnohočlen G(x) řádu *m*, který náleží k $e = 2^m - 1$. Generující mnohočlen je nerozložitelný a primitivní, což znamená, že je beze zbytku dělitelný pouze sám sebou a jedničkou. K detekci všech dvojitých chyb je nutné, aby G(x) nedělil beze zbytku $x^i + x^j$ , kde *i* a *j* může nabývat libovolné hodnoty z množiny *Z* (celých kladných čísel). Pokud $i < j$, můžeme $x^i + x^j$ vyjádřit jako $x^i . (1 - x^{j-i})$. Potom stačí, aby ($x^{j-i}$+1) nebyl dělitelný G(x). Jelikož $j-i < n \leq e$ a G(x) náleží exponentu *e*, nemůže G(x) dělit výraz ($x^{j-i}$+1) beze zbytku. To nám zaručuje detekci dvojnásobných chyb.

3) *Jakýkoliv cyklický kód vytvořený generujícím mnohočlenem stupně n-k detekuje jakýkoliv shluk chyb o délce n-k a menší.*

Polynom charakterizující chybu E(x) je možné převést do tvaru $x^i.E_1(x)$, který je řádu *b-1*. Takovýto shluk chyb bude detekován, pokud G(x) nedělí beze zbytku E(x), což může nastat pouze v případě, že současně nedělí beze zbytku $E_1(x)$. Pokud je splněna podmínka $b \leq n-k$, je G(x) vždy vyššího řádu než $E_1(x)$. Pokud dělíme číslo menší číslem větším, vždy dostaneme zbytek. Tento zbytek signalizuje chybu, která je tímto způsobem detekována.

4) *Procento shluků délky b > n-k, které zůstává nezjištěno je v případě b > n-k rovno $2^{-(n-k)}$.100 (%). Pokud je b = n-k, je procento nezjištěných shluků chyb $2^{-(n-k-1)}$.100(%).*

Chybový mnohočlen E(x) vyjádříme ve tvaru $x^i.E_1(x)$, kde $E_1(x)$ je řádu b-1. Polynom $E_1(x)$ pak určitě obsahuje členy $x^0$ a $x^{b-1}$, dále obsahuje b-2 členů $x^j$, kde $0<j<b-1$, které mají koeficient 0 nebo 1. Takže existuje $2^{b-2}$ rozdílných polynomů $E_1(x)$. Pokud $E_1(x)=G(x).M(x)$, nebude chyba detekována, $E_1(x)$ má totiž G(x) jako činitele. Je-li G(x) řádu *n-k*, musí být M(x) řádu *b-1*až (n-k). Pokud b-1=n-k, bude M(x)=1 a existuje pouze jedno $E_1(x)$, kdy vznikne nezjištěná chyba a to když se $E_1(x)=G(x)$. Poměr počtu nedetekovaných shluků k celkovému počtu možných shluké patřičné délky je $1/(2^{b-2})=2^{-(n-k-1)}$. Pokud b-1 > n-k, tak M(x) obsahuje členy $x^0$ a $x^{b-1-(n-k)}$ a má b-2 až (n-k) libovolných koeficientů. Poměr nedetekovaných chybových polynomů k celkovému počtu možných je $2^{b-b-(n-k)}/2^{b-2}=2^{-(n-k)}$.

5) *Jakýkoliv cyklický kód vytvořený generujícím polynomem $G(x)=(x^e+1).P_1(x)$ detekuje jakoukoliv kombinaci dvou shluků $E(x)=x^j.E_1(x)+ x^j.E_2(x)$.*

Pokud je $P_1(x)$ nerozložitelný a stupně alespoň takového, jako je délka kratšího shluku chyb. Dále pokud je *c+1* větší než součet délek obou shluků a délka kódu $n \leq c.e.$ (*e* je exponent, ke kterému náleží $P_1(x)$. Důkaz je uveden v [9].

# 2 Porovnání použitelných kódů

## 2.1 Cyklické kódy

Z množiny cyklických kódů je pro detekci dlouhých shluků chyb vhodný Fireův kód. Je to blokový cyklický kód určený vytvářecím mnohočlenem $G(x)=N(x) . (x^c + 1)$, kde N(x) je nerozložitelný mnohočlen řádu *m* a *c* není dělitelné *e*. Protože ho v tomto případě použijeme pouze k detekci chyb, tak je délka detekovatelného shluku chyb rovna počtu zabezpečovacích prvků, neboli řádu vytvářecího mnohočlenu. Jeho parametry vypočteme podle následujících vztahů:

Exponent *e*: $e = 2^m - 1$

Délka kódové kombinace: $n = e . c$

Počet zabezpečovacích prvků: $r = c + m$

Počet zabezpečovaných prvků $k = n - r$

Následující tabulka zobrazuje parametry Fireových kódů, jejichž řád nerozložitelného mnohočlenu náleží do intervalu 2 až 10.

**Tab. 2.1:** Přehled nezkrácených Fireových kódů a jejich vytvářecích mnohočlenů.

| m<br>- | e<br>- | n<br>[b] | k<br>[b] | r<br>[b] | N(x)<br>- | $(x^c + 1)$<br>- | G(x) = N(x) . $(x^c + 1)$<br>- | $B_1$<br>[b] |
|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 9 | 4 | 5 | $(x^2+x+1)$ | $(x^3+1)$ | $(x^6+x^4+ x^3+x^2+x+1)$ | 6 |
| 3 | 7 | 35 | 27 | 8 | $(x^3+x+1)$ | $(x^5+1)$ | $(x^8+x^6+ x^5+x^3+x+1)$ | 8 |
| 4 | 15 | 105 | 94 | 11 | $(x^4+x+1)$ | $(x^7+1)$ | $(x^{11}+x^8+ x^7+x^4+x+1)$ | 11 |
| 5 | 31 | 279 | 265 | 14 | $(x^5+x^2+1)$ | $(x^9+1)$ | $(x^{14}+x^{11}+ x^9+x^5+x^2+1)$ | 14 |
| 6 | 63 | 693 | 676 | 17 | $(x^6+x^3+1)$ | $(x^{11}+1)$ | $(x^{17}+x^{14}+ x^{11}+x^6+x^3+1)$ | 17 |
| 7 | 127 | 1 651 | 1 631 | 20 | $(x^7+x^3+ 1)$ | $(x^{13}+1)$ | $(x^{20}+x^{16}+ x^{13}+x^7+x^3+1)$ | 20 |
| 8 | 255 | 3 825 | 3 802 | 23 | $(x^8+x^4+ x^3+x+1)$ | $(x^{15}+1)$ | $(x^{23}+x^{19}+ x^{18}+x^{16}+x^{15}+x^8+x^4+x^3+x+1)$ | 23 |
| 9 | 511 | 8 687 | 8 661 | 26 | $(x^9+x^4+1)$ | $(x^{17}+1)$ | $(x^{26}+x^{21}+ x^{17}+x^9+x^4+1)$ | 26 |
| 10 | 1023 | 19 437 | 19 408 | 29 | $(x^{10}+x^9+ x^5+x+1)$ | $(x^{19}+1)$ | $(x^{29}+x^{28}+ x^{24}+x^{20}+x^{19}+x^{10}+x^9+x^5+x+1)$ | 29 |

Tmavě zvýrazněné řádky jsou ty varianty kódy, které splňují základní požadavek – detekovat shluk chyb o délce 22 bitů. $B_1$ je délka shluku chyb v bitech, které je kód schopen detekovat. V posledních třech řádcích tabulky je délka kódového slova delší, než je délka bezchybného intervalu. Mohlo by se stát, že během přenosu jednoho kódového slova dojde ke dvěma a více shlukovým chybám. V tomto případě již není zaručena jejich detekce.

Existuje však zkrácená varianta Fireova kódu. Princip zkracování délky kódové kombinace je jednoduchý. Na prvních i místech v mnohočlenu P(x) u členů nejvyššího řádu nezkrácené kódové kombinace předpokládáme nuly. Bylo rozhodnuto použít délku kódové kombinace 1000 bitů. Parametry zkrácených kódů jsou uvedeny v tabulce 2.2:

**Tab. 2.2:** Přehled zkrácených Fireových kódů a jejich parametry.

| n [b] | k [b] | r [b] | n* [b] | k* [b] | R - | i [b] |
|---|---|---|---|---|---|---|
| 3825 | 3802 | 23 | 1000 | 977 | 0,9770 | 2825 |
| 8687 | 8661 | 26 | 1000 | 974 | 0,9740 | 7687 |
| 19437 | 19408 | 29 | 1000 | 971 | 0,9710 | 18437 |

Sloupec *n** označuje délku zkrácené kódové kombinace, *k** analogicky počet zabezpečovaných prvků. Ve sloupci *i* je uvedena hodnota zkrácení a ve sloupci *R* je vypočtena hodnota redundance. Čím více se hodnota *R* blíží k 1, tím méně nadbytečné informace se přenáší.

## 2.2 Systém s bitovým prokládáním

Nejprve byl použit zcela jednoduchý blokový Hammingův kód. Popis tohoto jednoduchého kódování lze nalézt téměř v každé literatuře. Základním požadavkem při výběru vhodného kódu pro bitové prokládání bylo, aby měl srovnatelný informační poměr s kódem Fireovým a tím nedošlo k znevýhodnění této varianty ještě před vzájemným srovnáním. Tomuto požadavku vyhovoval z množiny Hammingových kódů jako první kód (128,121), který je schopen detekovat dvojnásobnou chybu. S použitím prokládací matice je možné tento kód použít k detekci zadaného shluku chyb. Dále byla ověřena možnost použití dvou Fireových kódů nižšího řádu.

**Hammingův kód:**
Délka shluku chyb je $b \leq 22$ bitů, Hammingův kód je schopen detekovat dvounásobnou chybu $t = 2$.
Počet řádků matice musí vyhovovat nerovnosti $j \geq (b / t)$ ... Minimální počet řádků matice je $j = 11$.
Počet sloupců matice odpovídá počtu znaků použitého kódu: $n = 128$.
Ověření splnění podmínek: $n \leq ((A + b) / j)$ -> $128 \leq ((1800+22)/11)$ -> $128 \leq 282$.
Pro srovnání byl testován ještě stejný Hammingův kód s použitím 12 řádkové matice.

**Fireův kód (279,265)**
Je schopen detekovat chybu o délce $t = 11$. Minimální počet řádků matice je $j = 2$, což vyhovuje nerovnosti $j \geq (b / t)$. Počet sloupců matice je 279.
Ověření splnění podmínek: $n \leq ((A + b) / j)$ -> $279 \leq ((1800+22)/2)$ -> $128 \leq 911$.

**Fireův kód (693,676)**
Je schopen detekovat chybu o délce $t = 14$. Minimální počet řádků matice je $j = 2$, což vyhovuje nerovnosti $j \geq (b / t)$. Počet sloupců matice je 693.
Ověření splnění podmínek: $n \leq ((A + b) / j)$ -> $693 \leq ((1800+22)/2)$ -> $693 \leq 911$.

Byl porovnán informační poměr a složitost realizace jednotlivých řešení.

**Srovnání kódů**

Pro porovnání hodnot bylo použito tohoto hodnocení. Kód, který je ve zkoumaném aspektu nejlepší, obdrží 0 bodů. Ostatní obdrží hodnocení, které vyjadřuje procentuální rozdíl mezi hodnotou hodnoceného kódu a kódu referenčního, který obdržel 0 bodů. Při použití tohoto typu hodnocení získává nejvhodnější kód nejmenší počet bodů. V následující tabulce není obsažena položka zpoždění, které vnáší kodér a dekodér do přenosové cesty. Z vlastností obou se ale dá vyvodit, že zpoždění u systémů s prokládáním je větší než u kodeku s cyklickým kódem.

V následující tabulce jsou přehledně uvedeny složitosti a redundance všech zkoumaných kódů.

**Tab. 2.3:** Výsledné srovnání všech kódů.

| Kód | n* [bit] | k* [bit] | b1 [bit] | redundance | redundance body | složitost [MU] | Složitost body | celkem bodů |
|---|---|---|---|---|---|---|---|---|
| Fireův (1000,977) | 1000 | 977 | 23 | 0,977 | 0 | 3000 | 0 | **0** |
| Fireův (1000,974) | 1000 | 974 | 26 | 0,974 | 0,13 | 3052 | 1,73 | **1,86** |
| Fireův (1000,971) | 1000 | 971 | 29 | 0,971 | 0,26 | 3058 | 1,93 | **2,19** |
| Kód | Rozměry matice | | b1 | redundance | redundance | složitost | složitost | celkem |
| | n | j | [bit] | | body | [MU] | body | bodů |
| Hammingův (128,121) | 128 | 11 | 22 | 0,945 | 1,38 | 3086 | 2,87 | **4,24** |
| Hammingův (128,121) | 128 | 12 | 21 | 0,945 | 1,38 | 3342 | 11,40 | **12,78** |
| Fireův (279,265) | 279 | 2 | 22 | 0,950 | 1,18 | 3209 | 6,97 | **8,15** |
| Fireův (693,676) | 693 | 2 | 28 | 0,975 | 0,07 | 5777 | 92,57 | **92,63** |

Pro porovnání byly stanoveny dva parametry. Složitost kodéru a dekodéru vyjádřená pomocí paměti, která je pro činnost kodeku nutná a vyjádření poměru redundantních a informačních bitů. Kodér Fireova kódu vyžaduje dva moduly vstupní paměti o *k** buňkách, složitost vlastního kodéru je pak rovna počtu zabezpečovacích prvků neboli řádu generujícího mnohočlenu navýšenému o hodnotu jedna. Lze snadno dokázat, že pro uložení mnohočlenu do paměti je potřeba *m+1* paměťových buněk. Složitost dekodéru je opět určena počtem *m+1* paměťových buněk v děličce a *n** buňkami, které tvoří vyrovnávací paměti dekodéru.

Složitost kodéru i dekodéru je při použití prokládání určena velikostí prokládací matice, počtem zabezpečovacích prvků a mechanismem vlastního kódování. Tedy dvojnásobku součinu *j . n* navýšenému o délku samotného kódového slova. V případě použití cyklického kódování je nutno připočíst ještě jeho vlastní paměťovou náročnost uvedenou v předchozím odstavci.

Z tabulky 2.3 vyplývá, že nejvhodnějším kódem je **Fireův kód (1000, 977)**, který získal v celkovém hodnocení 0 bodů. Pro tento kód byl proveden návrh kodéru a dekodéru.

# 3 Návrh kodeku

V první fázi bylo nutno vyřešit realizační strukturu, jakou bude kodek realizován.
Kodér i dekodér by v současné době bylo možno realizovat:

1) Pomocí logických obvodů, technologie TTL nebo CMOS.
   Již při prvotní úvaze bylo od tohoto řešení upuštěno, kvůli značné náročnosti návrhu a možnosti vzniku chyb během návrhu a vlastní realizace desky.
2) Pomocí programovatelného logického pole.
   Nebyl by nutný komplikovaný návrh desky plošných spojů a její výroba, je zde však komplikovaný návrh a implementace bloku řízení.
3) Čistě softwarové řešení pomocí osobního počítače.
   Nejsnazší varianta, ovšem závislá na použité platformě softwarového a hardwarového vybavení, málo univerzální.
4) Pomocí programovatelných logických polí řízených mikroprocesorem.
   Tato varianta poskytuje možnost případné změny kódovacího systému v budoucnu a usnadňuje návrh bloku řízení, který je realizován mikroprocesorem.

Vlastní návrh se obvykle sestává z těchto fází:
Nejprve se vytvoří specifikace, ve které se definuje funkční chování a požadavky kladené na navrhovaný systém. Potom se provádí návrh obvodu pomocí jazyka VHDL. Návrh se poté ověřuje v simulátoru, mezi nejznámější patří program ModelSim. Poté se ověří funkčnost navrženého systému a návrh v jazyce VHDL se předá k dalšímu zpracování, kde se provádí syntéza neboli generování logického schématu ve formě netlistu. Poté se aplikuje překlad a stanoví se omezující podmínky jako přiřazení pinů apod. Následuje rozdělení do logických bloků cílové platformy, mapování a rating (propojení jednotlivých bloků). Poté se provádí druhé testování. Provede se statická časová analýza, určí se tzv. kritická cesta a hledá se maximální frekvence hodinového signálu, se kterou je systém ještě funkční. Posledním krokem je vytvoření programovacího souboru, který se použije k naprogramování součástky.

Bylo však rozhodnuto provést pouze specifikaci, popis funkčního chování a simulaci funkčnosti kodéru a dekodéru. Hlavním důvodem tohoto rozhodnutí bylo zadání, které požaduje teoretický popis aniž by blíže specifikovalo konkrétní požadavky. Systém tak může být realizován jakýmkoliv výše zmíněným způsobem.

Pro ověření byla zvolena aplikace v jazyce Java. Dále byla zvolena vysoká úroveň abstrakce, při které se ověření neprovádí na bitové úrovni ale pomocí aritmetiky dvojkových čísel uložených v souborovém systému PC jako textově-datové soubory.
Při návrhu byly dále zohledněny tyto hlediska:

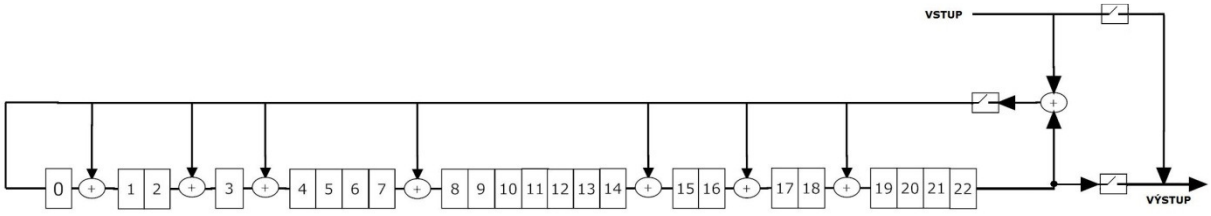
1) Jazyk Java byl vybrán pro svou jednoduchost, podobnost s jazykem C a také proto, že aplikace napsaná v prostředí Java poběží po kompilaci bez sebevětších potíží pod jakýmkoliv operačním systémem a na libovolné architektuře procesoru. Jazyk Java je totiž interpretovaný, místo strojového kódu se vytváří tzv. mezikód, což je jakýsi strojový kód pro virtuální stroj Javy. Před prvním provedením je dynamicky zkompilován do strojového kódu daného počítače – provádí se tzv. „just in time compilation“, která umožňuje rychlejší provádění instrukcí za běhu. Nevýhodou oproti ostatním jazykům, které provádějí statickou

kompilaci, je pomalejší start programu a u jednodušších programů i větší paměťová náročnost, protože spolu se samotnou aplikací je spuštěn i JVM.

2) Samotný program byl navrhován s ohledem na jednoduchost a přehlednost. Až v poslední fázi bylo pro pohodlnější obsluhu přidáno GUI – grafické uživatelské rozhraní. Není však nezbytné, celou simulaci lze provést pomocí příkazového řádku. Zdrojové soubory a zkompilované .jar archivy je možno nalézt na přiloženém CD v adresářích Simulace a Programy. Pro případ neexistence virtuálního stroje Javy na hostitelském počítači byl program také konvertován na spustitelný soubor systému MS Windows.

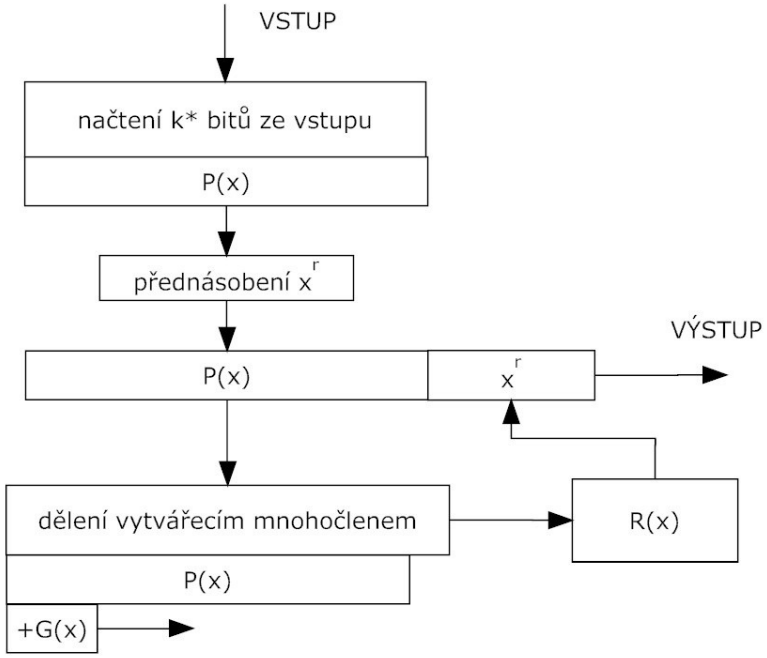
## 3.1 Návrh kodéru

Kodér realizuje Fireův kód (1000, 977). Jeho vytvářecí mnohočlen je zadán pomocí nerozložitelného polynomu osmého řádu ($x^8+x^4+x^3+x+1$), který vynásobíme ($x^c + 1$) - v tomto případě ($x^{15} + 1$).
Po vynásobení získáme $G(x) = (x^{23}+x^{19}+x^{18}+x^{16}+x^{15}+x^8+x^4+x^3+x+1)$. Základem dekodéru je struktura, která dělí vstupní polynom nezabezpečené bitové posloupnosti P(x) polynomem G(x). Z tohoto dělení nás zajímá zbytek po dělení, který tvoří vlastní zabezpečení informačních bitů a je zapsán do posledních *k** pozic kódového slova. Na následujícím obrázku je zobrazeno blokové schéma kodéru pro tento kód, které bývá zobrazováno ve většině publikací.




**Obr. 2:** Základní blokové schéma kodéru Fireova kódu (1000, 977).

Toto schéma je možná poněkud zavádějící, bylo proto zvoleno jiné vyjádření v podobě grafického vývojového diagramu. V následujícím diagramu není vyjádřen nadřazený blok řízení a vstupní a výstupní vyrovnávací paměti. V tomto systému je použito rozhodovací zpětné vazby, kde se zpětným kanálem přenáší pouze informace o tom, že v určitém kódovém slově byla detekována chyba a jednoznačné označení tohoto bloku dat. Systém je navržen typu nepřerušovaného ARQ (Automatic Repeat Request), což znamená, že se kódují a odesílají bloky dat ze vstupu bez přerušení do té doby, než zpětným kanálem přijde zpráva o detekované chybě. V případě signalizace chyby se dokončí kódování aktuálního slova a dojde k opětovnému zakódování a odeslání předchozí sekvence, která byla označena za chybnou. Při přenášení zpráv zpětným kanálem se neuvažuje možnost napadení těchto informací chybou, kanál je považován za spolehlivý.



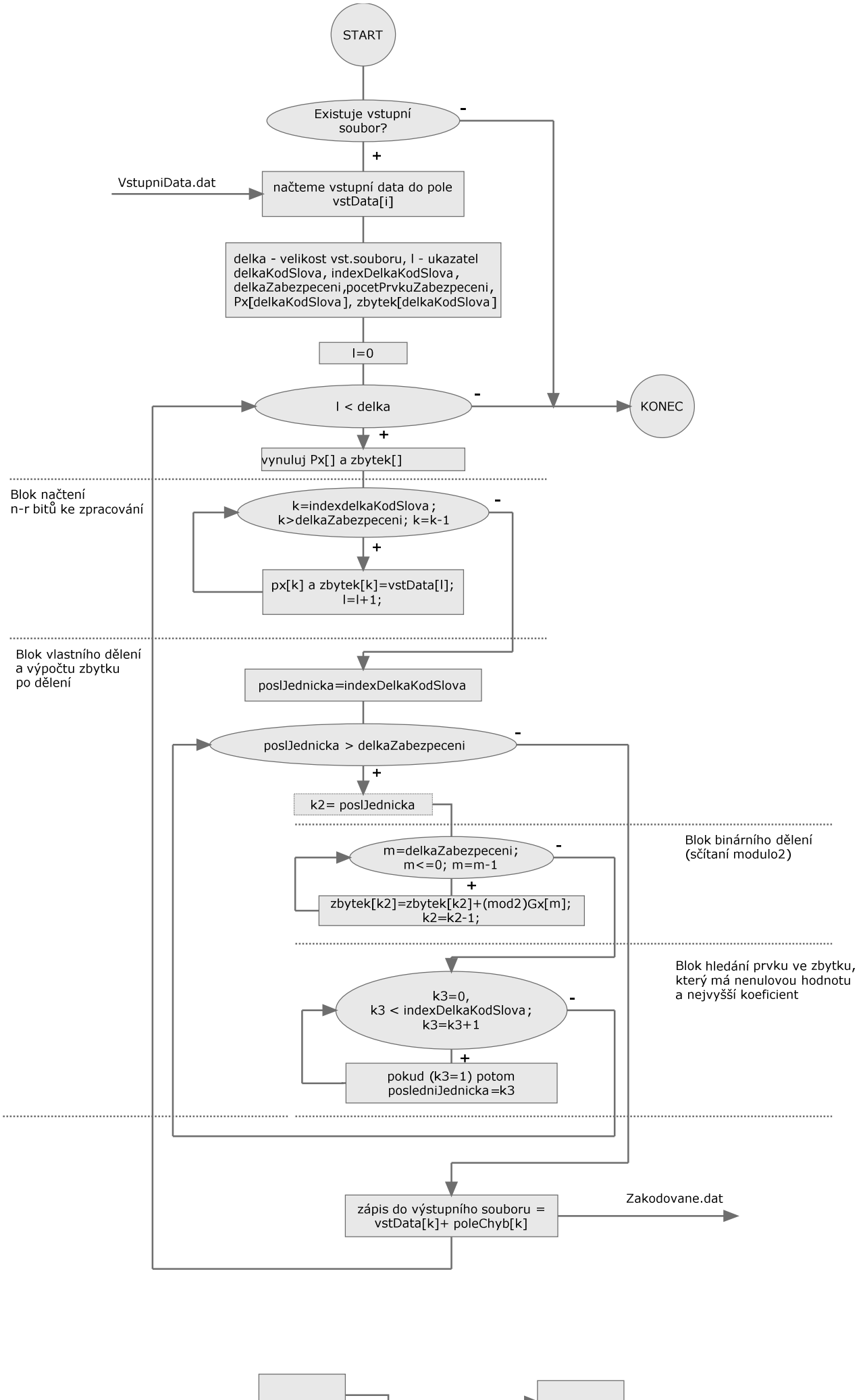

**Obr. 3:** Logické blokové schéma kodéru Fireova kódu (1000, 977).

Pomocí schématu na obrázku č. 3 můžeme přistoupit k návrhu vývojového diagramu kodéru, který bude interpretován pomocí programovacího jazyka Java.
Vývojový diagram kodéru Fireova kódu *(*1000, 977*)* je vyobrazen na obrázku č. 4. Diagram je zjednodušený, pro přehlednost nejsou obsaženy všechny metody a proměnné použité ve finálním zdrojovém kódu aplikace. Jedná se především o různá ošetření proti chybným vstupům.

**Zkrácení**
Použili jsme zkrácený kód (1000,977). Původní nezkrácený kód měl mít jinou délku kódového slova (3825,3802). Na tuto skutečnost musíme při návrhu pamatovat. Jakým způsobem se změní kodér a dekodér při použití zkrácených Fireových kódů? Na prvních *i* místech nezkrácené kódové kombinace předpokládáme nuly. Kodér je stejný pro zkrácený i pro nezkrácený kód, je určen přímo vytvářecím mnohočlenem G(x).

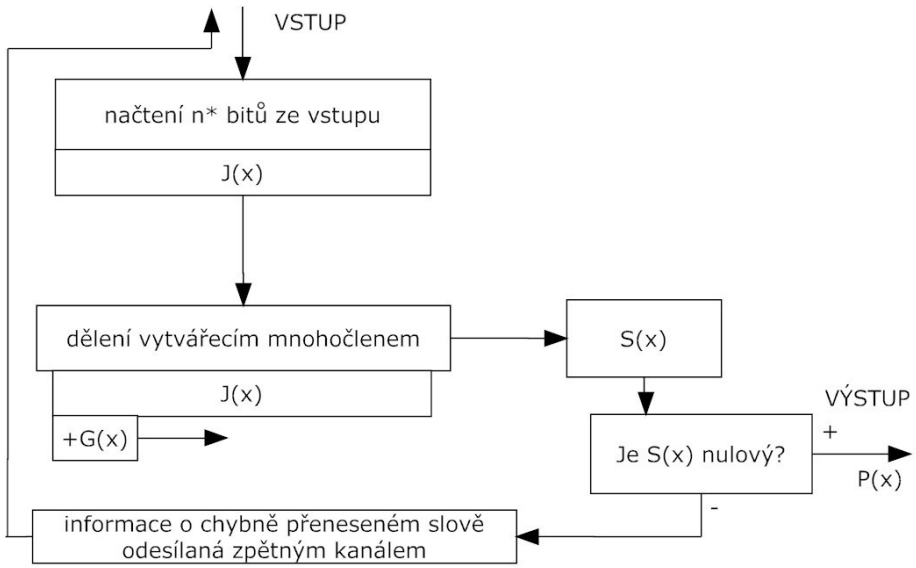
**Popis vývojového diagramu.**
V první části programu jsou po ověření načtena vstupní data ze souboru do proměnné *vstData*. Proměnná *vstData* představuje vlastní bitový proud nezabezpečené zprávy, který bylo nutné načtením do paměti připravit ke zpracování. V dalším bloku je do proměnné *Px* načteno *k** bitů ze vstupu a tato data jsou zdvojena, aby při dělení nedošlo k jejich změně. Kopie vstupního slova se uloží do proměnné *zbytek*. Další blok provádí opakovaný součet mod2 binárních dat uložených v proměné *Px* a *Gx* do té doby, než získáme zbytek. Poté je do výstupního souboru nejprve zapsána zabezpečovaná posloupnost *Px* a zbytek po dělení uložený v proměnné *poleChyb*. Proměnná *l* slouží pro adresování paměťových pozic v proměnné *vstData*. Poté co je vstupní slovo zakódováno a uloženo ( = odesláno), dojde k vynulování proměnných *Px* a *zbytek*. Je načtena nová sekvence a proces se opakuje do té doby, než se narazí na konec pole *vstData*. Zdrojový kód kodéru je uveden v příloze.

**Obr. 4:** Vývojový diagram kodéru Fireova kódu



## 3.2 Návrh dekodéru

Pro dekódování Fireova kódu byl použit upravený Meggitův dekodér, který uskutečňuje kontrolu správnosti přenosu. Výsledkem první fáze je mnohočlen syndromu *S(x)*. Je li roven nule, byl přenos bezchybný.




**Obr. 5:** Zjednodušené blokové schéma dekodéru Fireova kódu v detekčním režimu

**Zkrácení**

Dekodér zkráceného kódu však již musí reflektovat skutečnost, že v nejvyšších členech mnohočlenu přijaté zprávy předpokládáme nuly. Jsou možná dvě obvodová řešení:

1) Použijeme stejný dekodér jako pro nezkrácený kód a před přijetím zabezpečené posloupnosti k ní přidáme *i* nulových bitů.
2) Upravíme zapojení děličky mod G(x) dekodéru. Děličku mod G(x) s přednásobením $x^r$ je nutno doplnit o další vstupy do posuvného registru tak, aby to odpovídalo struktuře mnohočlenu f(x), což je zbytek po dělení $x^{i+r}$ s vytvářecím mnohočlenem G(x). Vlastní dělička je potom kombinací mnohočlenů f(x) a G(x).

Z důvodu úspory paměťového prostoru dekodéru bylo přistoupeno k druhé variantě. Pro uvažovaný Fireův kód je r = 23 a požadované zkrácení *i* = 3825-1000 = 2825. Zbytek po dělení f(x) získáme:

$$f(x) = \frac{x^{i+r}}{G(x)} = \frac{x^{2825+23}}{x^{23} + x^{19} + x^{18} + x^{16} + x^{15} + x^8 + x^4 + x^3 + x + 1}$$

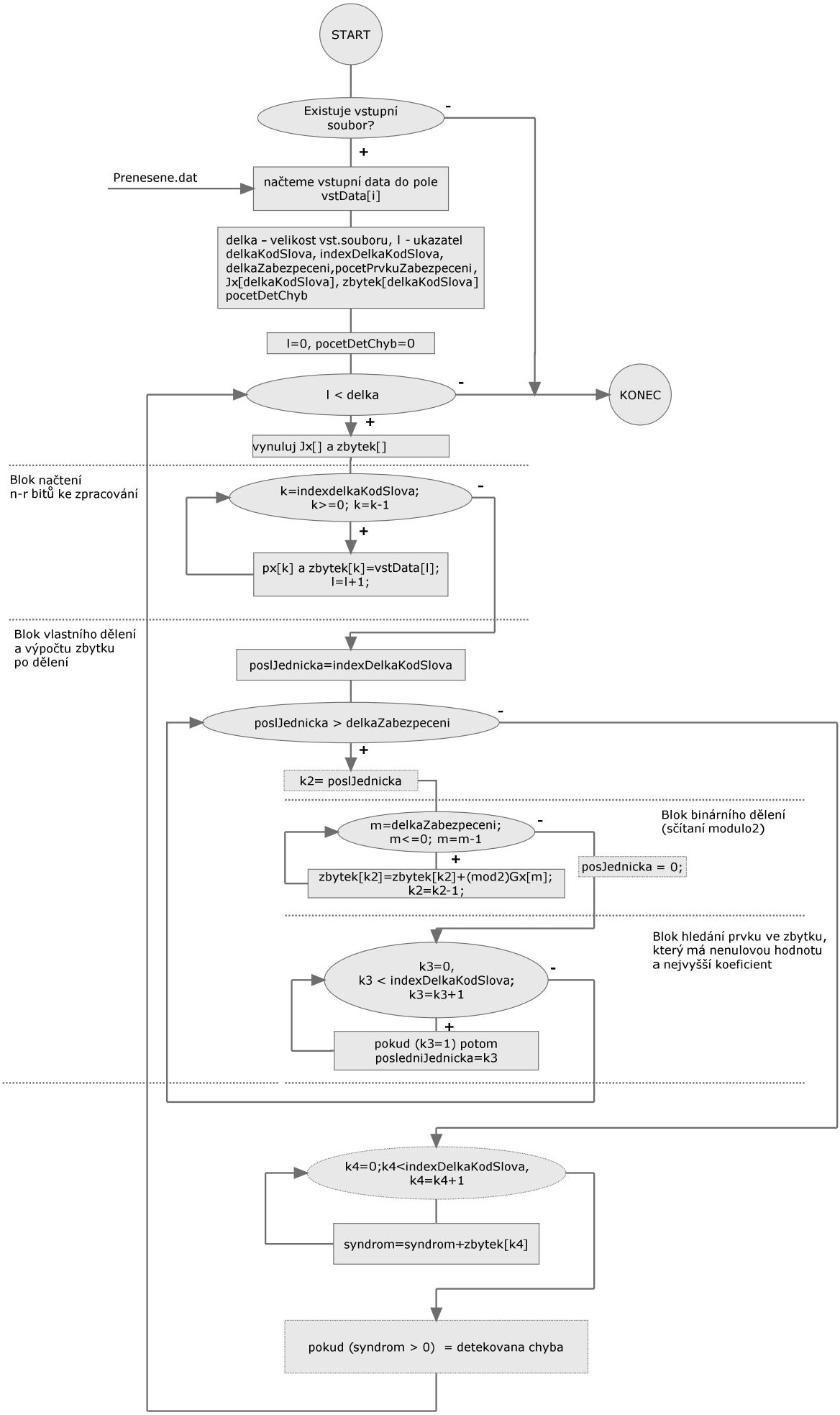
Pro výpočet tohoto výrazu byl napsán jednoduchý jednoúčelový program v jazyce Java, pomocí kterého byl určen zbytek po dělení. Program vznikl úpravou algoritmu používaného pro kódování a je možno jej nalézt na přiloženém CD v adresáři Programy pod jménem Vypocet.java.

Zbytek po dělení vypočtený pomocí programu je: $f(x) = x^{22}+x^{18}+x^{13}+x^{7}+x^{3}+1$.
Tím jsme dostali dva mnohočleny, které tvoří děličku dekodéru. Členy náležející pouze f(x) se účastní pouze první etapy (detekce) a ty, které jsou pouze v G(x) se účastní pouze druhé etapy (korekce). Mnohočleny obsažené v obou se použijí v obou fázích. První fáze probíhá jako přednásobení $x^{i+r}$ přes vazby určené mnohočlenem f(x), následuje dělení G(x) pro zisk syndromu.

Pro potřeby simulace však byla použita první varianta, tedy doplnění zakódované posloupnosti dat H(x) *i* nulovými bity. Podle schématu na obr. č. 5 byl vytvořen vývojový diagram dekodéru. Principielně je velmi podobný kodéru, obsahuje pouze několik rozdílných prvků. Dá se konstatovat, že je jednodušší než kodér. Je zobrazen na obr. č. 6.

**Vývojový diagram dekodéru.**
V první části programu jsou po ověření načtena vstupní data ze souboru do proměnné *vstData*. Proměnná *vstData* představuje vlastní bitový proud nezabezpečené zprávy, který bylo nutné načtením do paměti připravit ke zpracování. V dalším bloku je do proměnné *Px* načteno *k** bitů ze vstupu. Další blok provádí opakovaný součet mod2 binárních dat uložených v proměnné *Px* a *fx* do té doby, než získáme zbytek, který uložíme do proměnné syndrom. Poté je syndrom testován, je-li nulový. Pokud ano, přenos proběhl v pořádku a nenastala žádná chyba. Pokud má nenulovou hodnotu, nastala během přenosu chyba a je požádáno zpětným kanálem o opětovné vyslání dat. Poté dojde k vynulování proměnných *Px* a *zbytek*. Je načtena nová sekvence a proces se opakuje do té doby, než narazí na konec pole *vstData*. Nakonec se vypíše, v kolika případech nastala chyba. Zdrojový kód je uveden v příloze a na přiloženém CD-ROM.

**Obr. 6:** Vývojový diagram dekodéru Fireova kódu



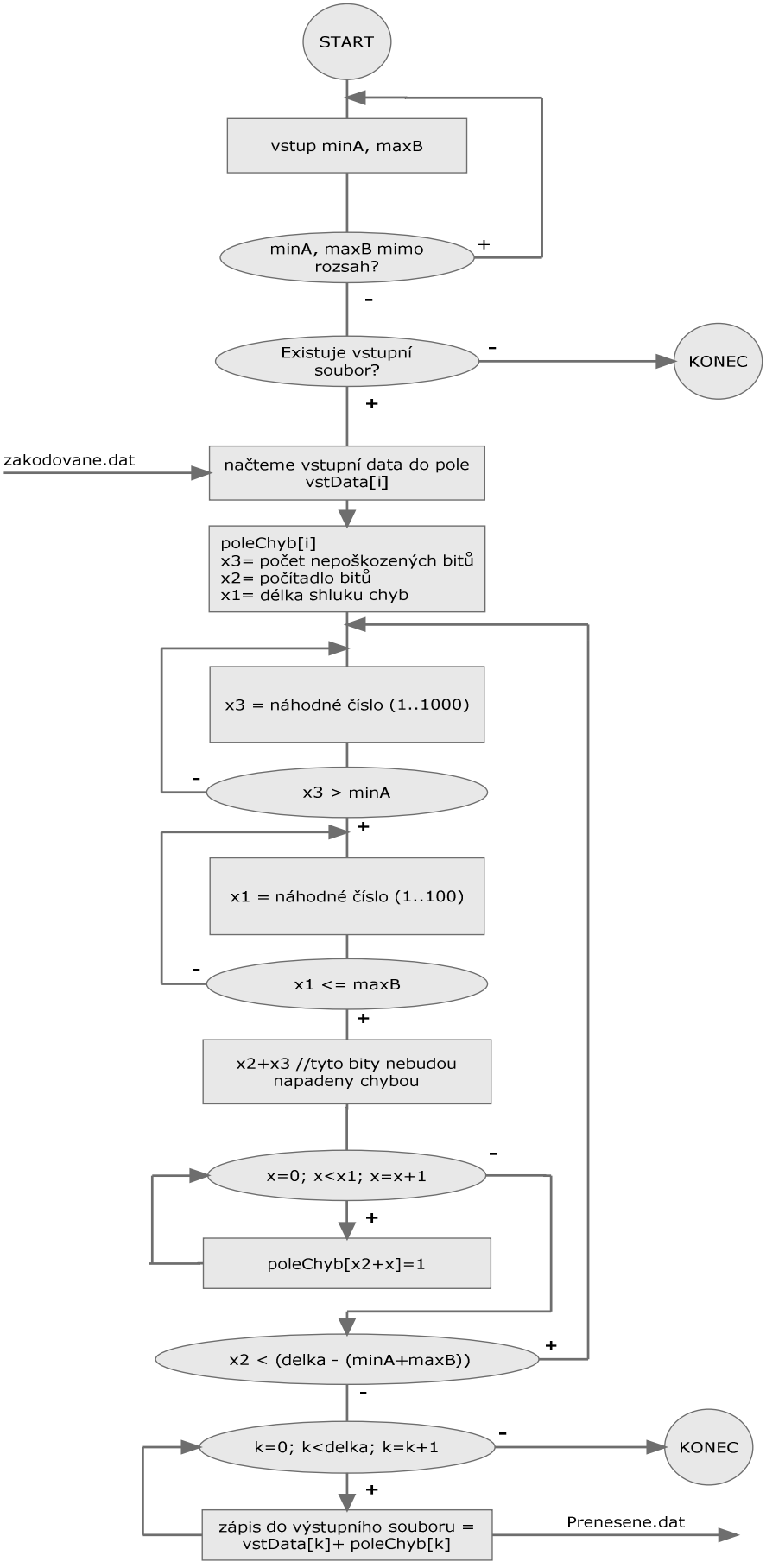
# 4 Simulace

Ve fázi kdy byl hotov kodér i dekodér přišla na řadu fáze testování, kdy byla ověřována funkčnost navrženého systému. Princip testování je jednoduchý. Vstupní data jsou pomocí kodéru zakódována a uložena do výstupního souboru. Potom je prováděna simulace přenosu, dle zadaných parametrů je generován chybový mnohočlen E(x), který je sečten s výstupním souborem kodéru a uložen jako vstupní soubor dekodéru. Zároveň je vypsána a uložena informace o počtu vzniknuvších chyb. Dekodér provede dekódování, detekci shluků chyb a vypíše informaci o počtu detekovaných chyb. Byla použita abstrakce na velmi vysoké úrovni, veškerá data jsou uložena v textových souborech uložených s příponou .dat. Textové vstupní testovací soubory byly pomocí jednoduchých skriptů naplněny náhodnými posloupnostmi nul a jedniček. Tyto textové soubory tvoří základní uživatelské rozhraní mezi logikou programu a souborovým systémem počítače, na kterém je simulace provozována. Jedničky a nuly jsou z textového formátu při načtení konvertovány do logických hodnot 0 a 1, nad kterými jsou potom prováděny veškeré operace kódování, dekódování a testování. Naopak při ukládání na výstup jsou z logického (numerického) tvaru převedeny na text. Pro účely simulace jsou operace prováděné nad souborovým systémem osobního počítače považovány za stoprocentně zabezpečené, souborový systém je považován za spolehlivý. Vznik možných aditivních chyb vzniklých při operacích zápisu a čtení z disku není uvažován.

## 4.1 Simulátor přenosu – generátor chyb

Na obrázku č. 7 je zobrazen vývojový diagram generátoru chybových mnohočlenů, který simuluje přenos dat diskrétním kanálem. Nejprve jsou načteny vstupní údaje o maximální délce shluku chyb a minimální délce bezchybného intervalu. Je provedeno ověření, zda byla zadána smysluplná hodnota a také, zda byla zadána hodnota z povoleného intervalu. Pro účely simulace byla povolena možnost zadat délku shluku chyb o délce 1 až 30, délka bezchybného intervalu může nabývat hodnot 1 až 2000. Při simulaci je samozřejmě nutno zadávat údaje shodné se zadáním, aplikace však umožňuje testovat i stavy s jinými hodnotami proměnných. Dále následuje test na existenci vstupního souboru a poté jsou načítána data z výstupního souboru kodéru *zakodovane.dat.* Následuje generování náhodných čísel. Nejprve je určována hodnota délky bezchybného intervalu, která může nabývat hodnot 1800 - 10 000. Poté je pomocí generování náhodného čísla určena hodnota délky shluku chyb, je však omezena intervalem 1 až 22. Tento algoritmus se neustále opakuje, dokud není chybový mnohočlen stejné velikosti jako vstupní data. Chybový mnohočlen je tvořen nulami, přerušovaný náhodně vygenerovanými shluky chyb. Poté je sečten chybový mnohočlen s mnohočlenem vstupních dat a výsledek je uložen do výstupního textového souboru – *prenesene.dat*

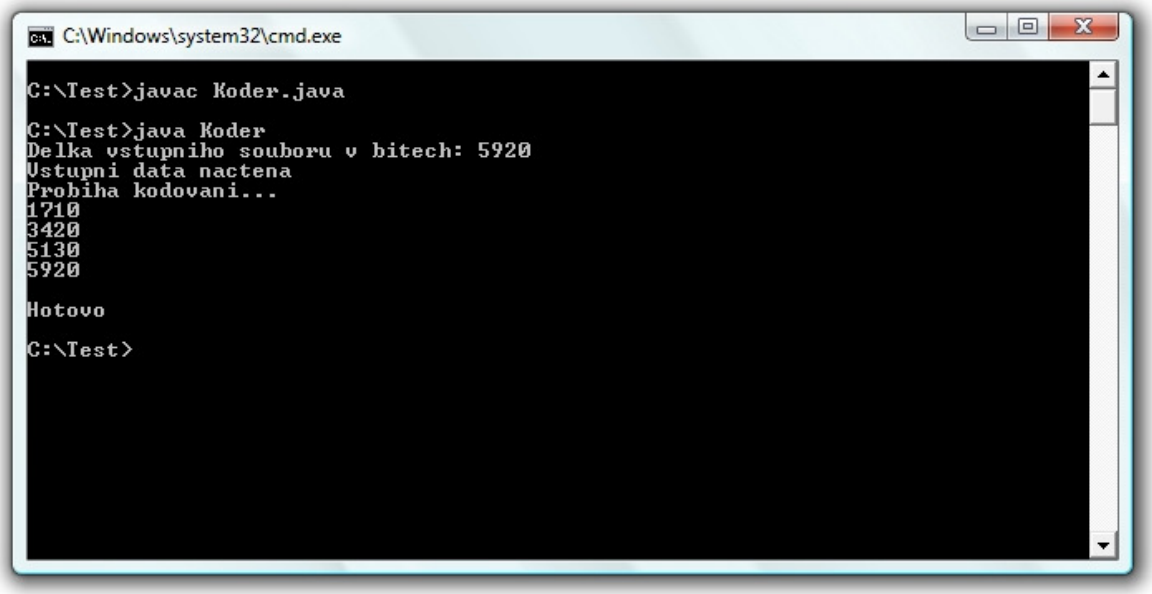
**Obr. 8:** Vývojový diagram generátoru chyb



## 4.2 Program Simulace přenosu dat

Na závěr byl vytvořen program, který spojuje všechny uvedené bloky do jednoho a umožňuje opakované kódování, přenos a dekódování za účelem ověření funkčnosti navrženého systému. Testování lze provádět dvěma způsoby.

1) Jednodušší metoda používá pouze příkazového řádku operačního systému. Výstup a veškerá komunikace s uživatelem probíhá v textové podobě. Jednotlivé bloky programu však v tomto případě nejsou spojeny a je nutno je volat z příkazového řádku jednotlivě. Programy jsou pojmenovány *Koder, Prenos a Dekoder.* Nevýhodou této metody je nemožnost opakovaného testování, kterou lze však snadno fixovat vytvořením dávkového souboru, který jednotlivé aplikace opakovaně spouští. Nelze také nastavovat parametry při generaci chybového mnohočlenu. Na obrázku č. 8 je snímek obrazovky zobrazující práci s programy pomocí příkazové řádky Windows.




**Obr. 9:** Práce s programem pomocí příkazové řádky

2) Programátorsky složitější avšak uživatelsky přívětivější je druhá metoda, používající grafického rozhraní. V tomto případě byly jednotlivé bloky zapouzdřeny do jednoho programu a jsou obsluhovány pomocí jednoduchého grafického rozhraní. Navíc byla do programu přímo přidána podpora opakovaného testování. Program se jmenuje *simulace.jar* a je možno ho nalézt na přiloženém CD v adresáři *Simulace*.

## 4.3 Popis programu a ovládání

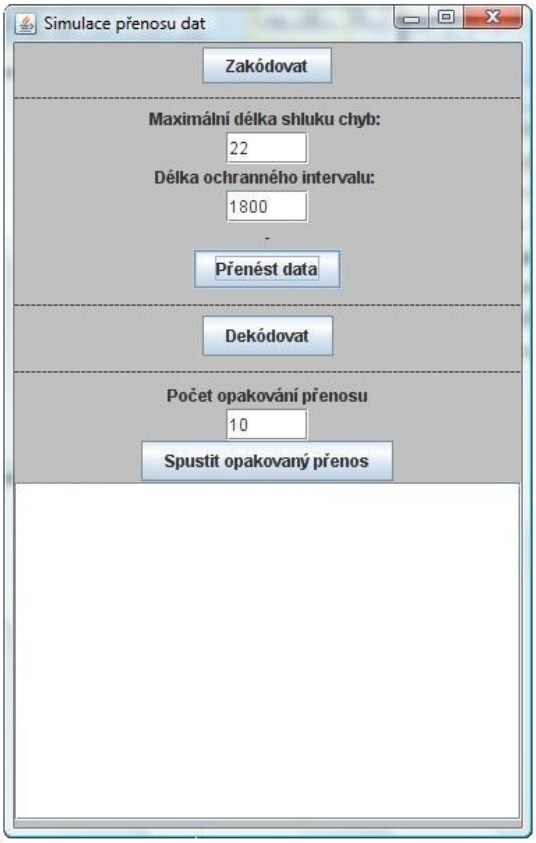
Ovládání je triviální, obsahuje pouze několik tlačítek, dvě vstupní pole a jedno textové pole výstupní. Přesto je vhodné ovládání a funkce pár slovy popsat. Program se spouští opět z příkazové řádky příkazem „java Simulace“ nebo lze poklepat ve Správci souborů na ikonu programu *Simulace.jar* a spustit program přímo. Snímek programu po spuštění zobrazuje obrázek č. 9.

Význam tlačítek není třeba dlouze vysvětlovat. Po klepnutí na tlačítko *Zakódovat*, dojde k načtení zadaného vstupního souboru, který se zakóduje a uloží do výstupního souboru *Zakodovane.dat.*

Druhý blok zobrazuje generátor chyb. Po zadání údajů (pro minimalizaci chyb a usnadnění práce jsou požadované hodnoty již předepsány) dojde k simulaci přenosu a poškození přenášených dat shluky chyb. Vstupní údaje lze samozřejmě měnit a tak je možno testovat chování kodeku, např. při zvýšení chybovosti nad mezní hodnotu apod.

Po klepnutí na tlačítko *Dekódovat* probíhá detekce chyb v „přeneseném“ proudu dat. Po skončení je do textové oblasti ve spodní části okna zobrazen výsledek dekódování.

Posledním blokem je blok opakovaného přenosu. Možnost opakování není omezena, avšak v případě nastavení počtu opakování většího než 50 je nutno počítat s průběhem testování v rámci hodin!




**Obr. 10:** Okno programu Simulace.

Jak vidno z obrázku č. 10 spodní textové pole slouží k informování uživatele o průběhu a výsledcích jednotlivých fází. Slouží také k zobrazování chybových výpisů v případě problémů s I/O diskovými operacemi. V případě opakovaného přenosu nejsou vypisovány všechny kroky, pouze výsledek. Je nutno vyčkat zobrazení výsledku, obzvlášť při nastavení většího počtu opakování a větší velikosti vstupního souboru.

Vývojový diagram celého programu není uveden. Je příliš složitý a pro pochopení funkce není podstatný. Program se skládá ze tří popsaných části kodéru, dekodéru a generátoru chyb. Zdrojový kód programu Simulace obsahuje navíc pouze vykreslení grafického rozhraní, ošetření chybových vstupů a některých hazardních stavů.


**Obr. 11:** Program Simulace - výstup

# 5 Výsledky simulací

Pomocí programu Simulace bylo prováděno opakované kódování, simulovaný přenos a dekódovaní testovaných vzorků dat. Nutno podotknout že v průběhu testování nebylo přihlíženo k zadané chybovosti kanálu BER. Vyjdeme li ze zadaných hodnot, dojde k napadení chybou u dvou bitů z milionu přijatých. Testování takto velkých bloků dat by zabralo příliš mnoho času, proto bylo testování prováděno při mnohonásobně větší chybovosti, řádově $2.10^{-3}$. Byla pouze dodržena ochranná vzdálenost 1800 bitů a maximální délka shluku chyb. Výsledky všech testů tvoří výpisy vygenerované programem.

---

```
KÓDOVÁNÍ:
Vstup: VstupniData2.dat – 239761 bitů.
Výstup: Zakodovane.dat – 246000 bitů.
Kódování úspěšně dokončeno.

SIMULACE PŘENOSU:
Vstup: Zakodovane.dat – 246000 bitů.
Výstup: Prenesene.dat – 246000 bitů.
Nastalo 40 chyb.

DEKÓDOVÁNÍ:
Vstup: Prenesene.dat – 246000 bitů.
Dekódováno. Počet detekovaných chyb :40
```

---

```
OPAKOVANE TESTOVANI
Vstup: VstupniData2.dat – 239761 bitů.
Data byla přenesena 5x.
Nastalo 257 chyb, bylo detekováno 257 chyb.
```

---

Poté byla testována varianta, kdy není dodržen ochranný interval A. Z výsledků simulace níže vyplývá, že v tomto případě došlo v několika případech ke dvěma nebo více chybám v jednom kódovém slově. Vstupní slovo bylo sice na dekodéru vyhodnoceno jako chybné, nelze už však specifikovat kolik chyb a jakých délek nastalo. V tomto případě již nelze dekódování považovat za 100% spolehlivé. Přesnou hodnotu by bylo možno vyhodnotit po pečlivém testování a zkoumání vstupních a výstupních dat. Pravděpodobnost, že k podobné situaci dojde v reálném kanále dle zadání je téměř nulová, vzhledem k malé hodnotě chybovosti kanálu.

```
KÓDOVÁNÍ:
Vstup: VstupniData4.dat - 238726 bitů.
Výstup: Zakodovane.dat - 245000 bitů.
Kódování úspěšně dokončeno.

SIMULACE PŘENOSU:
Vstup: Zakodovane.dat - 245000 bitů.
Výstup: Prenesene.dat - 245000 bitů.
Nastalo 43 chyb.

DEKÓDOVÁNÍ:
Vstup: Prenesene.dat - 245000 bitů.
Dekódováno. Počet detekovaných chyb :40
```

---

Další testování bylo již v pořádku.

---

```
OPAKOVANE TESTOVANI
Vstup: VstupniData2.dat –514286 bitů.
Data byla přenesena 5x.
Nastalo 534 chyb, bylo detekováno 534 chyb.
```

---

```
OPAKOVANE TESTOVANI
Vstup: VstupniData3.dat – 5920 bitů.
Data byla přenesena 20x.
Nastalo 54 chyb, bylo detekováno 54 chyb.
```

---

```
OPAKOVANE TESTOVANI
Vstup: VstupniData4.dat – 238726 bitů.
Data byla přenesena 25x.
Nastalo 1011 chyb, bylo detekováno 1011 chyb.
```

---

Dále bylo otestováno chování systému při maximální délce shluků chyb rovnou 30. Zde již systém nepracuje spolehlivě, jednou najde chyb více, podruhé méně.

```
KÓDOVÁNÍ:
Vstup: VstupniData4.dat - 238726 bitů.
Výstup: Zakodovane.dat - 245000 bitů.
Kódování úspěšně dokončeno.

SIMULACE PŘENOSU:
Vstup: Zakodovane.dat - 245000 bitů.
Výstup: Prenesene.dat - 245000 bitů.
Nastalo 35 chyb.

DEKÓDOVÁNÍ:
Vstup: Prenesene.dat - 245000 bitů.
Dekódováno. Počet detekovaných chyb :37
```

---

# 6 Závěr

Podařilo se vybrat vhodný kód a navrhnout principielní schéma kodeku, který zabezpečuje digitální přenos dat proti shlukujícím se chybám. Nejprve byla provedena teoretická rozvaha o možnostech a používaných typech kódování. Z množiny kódů byly vybrány ty, které jsou schopny detekovat shlukové chyby. Poté bylo provedeno srovnání vlastností jednotlivých kódů, zejména pak informační poměr jednotlivých kódů a složitost realizace vyjádřená počtem paměťových buněk. Byly testovány Fireovy kódy, bitové prokládání s Hammingovým kódem a bitové prokládání s kódy Fireovými. Kód s nejmenším počtem redundantní informace a s nejjednodušší realizací je Fireův kód (1000, 977).

Pro tento kód byly pomocí blokových schémat vytvořeny návrhy kodéru a dekodéru, které byly realizovány pomocí programovacího jazyka Java. Protože kód používá algebry mnohočlenů a binárního součtu modulo2, bylo nutné tyto funkce do návrhu zapracovat. V Javě byly vytvořeny i pomocné programy, např. generátor vstupních dat, který generuje soubor s náhodnou sekvencí nul a jedniček. Výsledkem celé práce pak je vytvoření simulačního programu, který spojuje bloky kódování, simulaci vzniku chyby během přenosu a dekódování (detekci) chyb v jeden celek.

Samotný návrh programu byl prováděn pomocí textového editoru PSPad, verze 4.5.3, který umožnuje barevně zvýraznit syntaxi programovacího jazyka. Kompilace programu byla prováděna s vestavěným Java kompilerem, který je součástí vývojového balíku Java Development Kit – JDK verze 1.6.0.6. Pro optimalizaci kódu a hledání chyb pomocí debuggeru byl použit program NetBeans IDE 6.0.1. Během vytváření tohoto programu jsem narazil na několik problémů. Jedním z nich je fakt, že pro uložení vytvářecího mnohočlenu 23. řádu do paměti je potřeba 24 paměťových buněk. Dalším, dá se říct největším problémem je číslování indexů datových polí v Javě. Máme pole délky n, první prvek pole má ale index 0 a poslední n-1. Bylo proto nutno spolehlivě vyřešit převody indexů při provádění jednotlivých operací. Program byl navrhován s použitím jednoduchého Fireova kódu (35,27) u kterého bylo možno výsledky kódování a dekódování ručně početně ověřit. Po otestování chování a po otestování správnosti interpretovaných výsledků byl požit vybraný kód vyššího řádu. Kodér i dekodér fungují korektně, v rámci parametrů zadání je zaručena 100% detekce shluků chyb. Systém byl testován i pro shluky chyb délky větší než 23, zde však již není zaručena detekce všech shluků. Paradoxně při překročení parametrů délky shuku chyb dochází občas k detekci více chyb, než jich bylo generováno. Nebylo však cílem práce zkoumat chování kodeku při překročení limitních podmínek. Cílem bylo nalézt funkční řešení zadaného úkolu.

Za zadaných podmínek, tj. pokud je délka shluku chyb maximálně 22 bitů a délka bezchybného interval 1800 bitů vykazuje kodek 100% účinnost při detekci chybových mnohočlenů.

# 7 Seznamy

## 7.1 Seznam použitých zkratek a symbolů

| | |
|---|---|
| k | délka vstupní informace určené k zakódování (v bitech) |
| n | počet bitů zakódované zprávy |
| r | počet zabezpečujících bitů |
| b | délka shluku chyb |
| P(x) | je informační mnohočlen nezabezpečené zprávy, vlastní informace. |
| G(x) | je generující mnohočlen určený zabezpečovacím kódem, stupně n-k. |
| M(x) | je mnohočlen podílu. |
| R(x) | je mnohočlen zbytku po dělení. |
| F(x) | je zakódovaný informační mnohočlen připravený na přenos kanálem. |
| E(x) | reprezentuje chybový mnohočlen, za ideálních podmínek by byl nulový. |
| H(x) | je přijatý zakódovaný mnohočlen ovlivněný chybou |
| S(x) | mnohočlen syndromu |
| $d_{min}$ | Hammingova vzdálenost |
| b1 | délka shluku chyb, který je kód schopen zabezpečit |
| MU | Memory Units – vlastní jednotka, navržená pro vyjádření paměťové náročnosti kodérů |
| VHDL | VHSIC Hardware Description Language - programovací jazyk sloužící pro popis hardware |
| JVM | Java Virtual Machine – virtuální stroj Javy |
| GUI | grafické rozhraní aplikací |

## 7.2 Seznam obrázků

## 7.3 Seznam tabulek

# 8 Použitá literatura


[1] HOUGHTON,A. *Error Coding for Engineers*. Kluwer academic Publishers. Boston, Dordrecht, London, 2001.

[2] VLČEK, Karel. *Komprese a kódová zabezpečení v multimediálních komunikacích*. Ben - technická literatura, 2. vyd. Praha, 2004. ISBN 80-7300-134-9.

[3] PUŽMAN, J. *Dálkový přenos dat*. SNTL/ALFA Praha, 1985

[4] NĚMEC, K. *Datová komunikace*, FEI VUT. Brno. 1998. 115 s

[5] KUBÁTOVÁ, Marie. *Diagnostika a spolehlivost*. 2006 [cit. 2008-04-30]. Dostupný z WWW: <http://cs.felk.cvut.cz/~kubatova/WEB/>.

[6] POLÁK, Karel. *Adaptivní kodek využívající Fireova kódu*. *Elektrorevue* [online]. 2002, č. 36 [cit. 2008-04-30]. Dostupný z WWW:<http://www.elektrorevue.cz/clanky/02036/index.html>

[7] KEOGH, James. *Java*. Translation: Ivo Fořt. Brno : CP Books, a.s., 2005. 275 s. ISBN 80-251-0839-3.

[8] DUŠEK, Josef. *Návrh a implementace opravných kódù pro systém Orpheus*. [s.l.], 2007. 60 s. České vysoké učení technické v Praze, Fakulta elektrotechnická. Vedoucí diplomové práce Ing. Martin Daněk, Ph.D.

[9] DRINČEV, P. *Detekční schopnosti dvojkových cyklických kódů*. Slaboproudý obzor 34, roč. 1973, číslo 2, str. 55 - 61.

# 9 Přílohy

V příloze je uveden zdrojový text programu, který simuluje funkce kodéru, dekodéru, přenos a vznik chyb. Naleznete také na přiloženém CD – *Simulace/Simulace.java*.

```
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Simulace {
          public static void main (String[] arg) {
          Okno okno = new Okno();
          }
      }

class Okno extends JFrame  {
JButton tl1 = new JButton("Zakódovat");
JButton tl2 = new JButton("Přenést data");
JButton tl3 = new JButton("Dekódovat");
JButton tl4 = new JButton("Spustit opakovaný přenos");
JTextArea to = new JTextArea("", 15, 30);
JTextField textob1 = new JTextField("22", 5);
JTextField textob2 = new JTextField("1800",5);
JTextField textob3 = new JTextField("10", 5);
JLabel pop1 = new JLabel("-------------------------------------------------
----------------------------------------");
JLabel pop2 = new JLabel("Maximální délka shluku chyb: ");
JLabel pop3 = new JLabel("Délka ochranného intervalu: ");
JLabel pop4 = new JLabel("-------------------------------------------------
----------------------------------------");
JLabel pop5 = new JLabel("-------------------------------------------------
----------------------------------------");
JLabel pop6 = new JLabel("Počet opakování přenosu");
JLabel popx = new JLabel("-");
JScrollPane scroll = new JScrollPane(to,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

// BLOK VYBERU A ZADANI PARAMETRU KODU
int delkaKodSlova=1000, delkaZabezpeceni=23;
int indexDelkaKodSlova=delkaKodSlova-1,
pocetPrvkuZabezpeceni=delkaZabezpeceni+1;
int Gx[] = {1,0,0,0,1,1,0,1,1,0,0,0,0,0,0,1,0,0,0,1,1,0,1,1};



public Okno ()
{
//definovani grafickoho vyhledu programu
super ("Simulace přenosu dat");
setSize(380, 600);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
Container kon = getContentPane();
kon.setBackground(Color.lightGray);
GridBagLayout srg = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
kon.setLayout(srg);

gbc.fill = GridBagConstraints.VERTICAL;
gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=0;
kon.add(tl1, gbc);

gbc.fill = GridBagConstraints.VERTICAL;
gbc.weightx = 0.5;
gbc.gridx=0;
```

```
gbc.gridy=1;
kon.add(pop1, gbc);

gbc.weightx = 0.5;
gbc.weighty = 0.2;
gbc.gridx=0;
gbc.gridy=2;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(pop2, gbc);

gbc.weightx = 0.5;
gbc.weighty = 0.2;
gbc.gridx=0;
gbc.gridy=3;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(textob1, gbc);

gbc.weightx = 0.5;
gbc.weighty = 0.2;
gbc.gridx=0;
gbc.gridy=4;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(pop3, gbc);

gbc.weightx = 0.5;
gbc.weighty = 0.2;
gbc.gridx=0;
gbc.gridy=5;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(textob2, gbc);

gbc.weightx = 0.0;
gbc.gridx=0;
gbc.gridy=6;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(popx, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=7;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(tl2, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=8;
kon.add(pop4, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=9;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(tl3, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=10;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(pop5, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=11;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(pop6, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=12;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(textob3, gbc);
```

```
gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=13;
gbc.fill = GridBagConstraints.VERTICAL;
kon.add(tl4, gbc);

gbc.weightx = 0.5;
gbc.gridx=0;
gbc.gridy=14;

gbc.fill = GridBagConstraints.HORIZONTAL;
kon.add(scroll, gbc);

setContentPane(kon);

tl1.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent evt)
      {
      File vstSoubor = new File("VstupniData4.dat");
      File vystSoubor = new File("Zakodovane.dat");
      int delka=(int)vstSoubor.length();
      to.setText(to.getText()+"KÓDOVÁNÍ:"+'\n');
      to.setText(to.getText()+"Vstup: "+vstSoubor+" - "+delka+"
bitů."+'\n');
      long a=zakoduj(vstSoubor,vystSoubor);
      if (a>0)
            {
            to.setText(to.getText()+"Výstup: "+vystSoubor+" - "+a+"
bitů."+'\n'+"Kódování úspěšně dokončeno."+'\n'+'\n');
            }

      }
         });
tl2.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent evt) {
    File vstSoubor = new File("Zakodovane.dat");
    File vystSoubor = new File("Prenesene.dat");
    int delka=(int)vstSoubor.length();
    to.setText(to.getText()+"SIMULACE PŘENOSU:"+'\n');
    to.setText(to.getText()+"Vstup: "+vstSoubor+" - "+delka+" bitů."+'\n');
    int b=prenes(vstSoubor,vystSoubor);
    if (b>0)
            {
            to.setText(to.getText()+"Výstup: "+vystSoubor+" -
"+vystSoubor.length()+" bitů."+'\n');
            to.setText(to.getText()+"Nastalo "+b+" chyb."+'\n'+'\n');
            }
    }
         });
tl3.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent evt) {
    File vstSoubor = new File("Prenesene.dat");
    to.setText(to.getText()+"DEKÓDOVÁNÍ:"+'\n');
    to.setText(to.getText()+"Vstup: "+vstSoubor+" - "+vstSoubor.length()+"
bitů."+'\n');
    int c=dekoduj(vstSoubor);
    if (c>0)
            {
            to.setText(to.getText()+"Dekódováno. Počet detekovaných chyb
:"+c+'\n'+'\n');
            }
    if (c==0)  {
               to.setText(to.getText()+"Nebyla detekována žádná chyba.");
              }
    }


         });
tl4.addActionListener(new ActionListener () {
    public void actionPerformed(ActionEvent evt) {
```

```
    to.setText(to.getText()+"OPAKOVANE TESTOVANI"+'\n');
    File vstSoubor1 = new File("VstupniData1.dat");
    File vystSoubor1 = new File("Zakodovane.dat");
    File vstSoubor2 = new File("Zakodovane.dat");
    File vystSoubor2 = new File("Prenesene.dat");
    File vstSoubor3 = new File("Prenesene.dat");
    int pocOpak=Integer.parseInt(textob3.getText());
    int celkemChyb=0;int celkemDetChyb=0;
    for (int i=0; i<=pocOpak;i++)
          {
            zakoduj(vstSoubor1,vystSoubor1);
            celkemChyb += prenes(vstSoubor2,vystSoubor2);
            celkemDetChyb += dekoduj(vstSoubor3);
          }
    to.setText(to.getText()+"Vstup: "+vstSoubor1+" -
"+vstSoubor1.length()+" bitů."+'\n');
    to.setText(to.getText()+"Data byla přenesena
"+pocOpak+"x."+'\n'+"Nastalo "+celkemChyb+" chyb, bylo detekováno
"+celkemDetChyb+" chyb."+'\n');

    }


         });

}

//---------------------------------------------------------------------------

//---------------------------------------------------------------------------

public long zakoduj(File vstSoubor,File vystSoubor){
//BLOK NACTENI VSTUPNICH DAT

String radek = new String("");
int delka=(int)vstSoubor.length();
int vstData[] = new int[delka] ;

if (delka == 0) //osetreni proti neexistujicim souborum
    {
    JOptionPane.showMessageDialog(null,"Vstupní soubor neexistuje!",
"Chyba",JOptionPane.PLAIN_MESSAGE);
    return -1;  //vyskoci z metody, je li chyba
    }
try
    {
    BufferedReader vstup = new BufferedReader(new FileReader(vstSoubor));
    radek=vstup.readLine();
    vstup.close();
    }   catch (IOException v)
        {
        to.setText(to.getText()+v+'\n');
        }

for (int i=0; i<delka;i++) //Prevod nactenych vstupnich dat do pole cisel
    {
    if  (radek.charAt(i) == '0')        {vstData[i]=0;}
    else if (radek.charAt(i) == '1')    {vstData[i]=1; }
    else  {
          JOptionPane.showMessageDialog(null,"Vstupní data jsou v
nesprávném formátu!", "Chyba",JOptionPane.PLAIN_MESSAGE);
          return -1;
          }
    }


if (vystSoubor.exists()) {vystSoubor.delete();} //ochrana proti neustalemu
zvetsovani souboru
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK VLASTNIHO KODOVANI

int l=0;
int Px[] = new int[delkaKodSlova];
int zbytek[] = new int[delkaKodSlova];

while (l<(delka ))  //nacteni dat
      {
      Arrays.fill(Px, 0);
      Arrays.fill(zbytek, 0);
      for (int k = indexDelkaKodSlova;k >= delkaZabezpeceni;k--)
          {
          if (l<delka){
                      Px[k]=vstData[l];
                      zbytek[k]=vstData[l];
                      l++;

                      }
          }


      int poslJednicka = indexDelkaKodSlova;
      while (poslJednicka > delkaZabezpeceni)  //vlastni scitani, XOR
            {
            int k2=poslJednicka;

            for (int m = delkaZabezpeceni;m >= 0;m--)
                {
                zbytek[k2] = zbytek[k2] ^ Gx[m];
                k2--;
                }
            poslJednicka=0;
            for (int k3=0;k3 < indexDelkaKodSlova;k3++)
                {
                if (zbytek[k3]==1) {poslJednicka=k3;}
                }
            }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK VYPISU NA VYSTUP
try   {
      PrintWriter vystup = new PrintWriter(new FileOutputStream(vystSoubor,
true));
      for (int k4=indexDelkaKodSlova;k4>delkaZabezpeceni;k4--)
          {
          vystup.print(Px[k4]);
          }
      for (int k5=delkaZabezpeceni;k5 >= 0;k5--)
          {
          vystup.print(zbytek[k5]);
          }
      vystup.close();

} catch (IOException v)
        {
        to.setText(to.getText()+v+'\n');
        }

}

return  vystSoubor.length();
 }
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------

public int prenes(File vstSoubor,File vystSoubor)
{
// BLOK OSETRENI VSTUPNICH PARAMETRU
```

```
int maxB=Integer.parseInt(textob1.getText()); //zisk vstupnich udaju
int minA=Integer.parseInt(textob2.getText());
if (maxB<1 || maxB>25 || minA<1 || minA>2000) //osetreni proti hodnotam
mimo rozsah
    {
    JOptionPane.showMessageDialog(null,"Vložená hodnota(y) jsou mimo
povolený rozsah!", "Chyba",JOptionPane.PLAIN_MESSAGE);
    return -1;
    }


//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK NACITANI DAT

String radek = new String("");
int delka=(int)vstSoubor.length();
int vstData[] = new int[delka];
if (delka == 0) //osetreni proti neexistujicim souborum
    {
    JOptionPane.showMessageDialog(null,"Vstupní soubor neexistuje!",
"Chyba",JOptionPane.PLAIN_MESSAGE);
    return -1;  //vyskoci z metody, je li chyba
    }

 try
    {
    BufferedReader vstup = new BufferedReader(new FileReader(vstSoubor));
    radek=vstup.readLine();
    vstup.close();
    }   catch (IOException v)
        {         to.setText(to.getText()+v+'\n');        }

    for (int i=0; i<vstSoubor.length();i++)
      {
      if (radek.charAt(i) == '0')
        {         vstData[i]=0;        }
      else vstData[i]=1;
      }  // mame nactena vstupni data

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK VYTVORENI CHYBOVEHO POLE a jeho inicializace
int poleChyb[] = new int[delka];
Arrays.fill(poleChyb, 0);

int x1=0,x2=0,x3=0, pocitadloChyb=0;
while (x2 < (delka - (minA + maxB+1000) ) )
  {
  //generovani chyboveho pole
  do {
  x3=(int)(Math.random()*10000);
  } while(x3<minA);

  x2+=x3;

  do {
  x1=(int)(2+Math.random()*10+Math.random()*10);
  }  while (x1>maxB);

  if ((x2+x1)<delka)  {
                      for (int x=0;x<x1;x++)
                          {
                          poleChyb[x2+x]=1;
                          }

                      }

  x2=x2+x1;


  }
```

```
for (int k8=0;k8 < delka;k8++)    {
      if ((poleChyb[k8]==1) && poleChyb[k8-1]==0 )
              {
              pocitadloChyb++;
              }
      }


//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK ZAPISU DO VYSTUPNIHO SOUBORU, SECTENI VSTUPU A CHYBOVEHO POLYNOMU
try {
PrintWriter vystup = new PrintWriter(new FileOutputStream(vystSoubor,
false));
    for (int k2=0;k2 < delka;k2++)
      {
      vystup.print(poleChyb[k2]^vstData[k2]);
      }
    vystup.close();

} catch (IOException v)
        {
        to.setText(to.getText()+v+'\n');
        }
return pocitadloChyb;




    }

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

public int dekoduj(File vstSoubor){
// BLOK NACITANI DAT a vstupni ochrany

String radek = new String("");
int delka=(int)vstSoubor.length();
int vstData[] = new int[delka] ;

if  (delka == 0) //osetreni proti neexistujicim souborum
    {
    JOptionPane.showMessageDialog(null,"Vstupní soubor neexistuje!",
"Chyba",JOptionPane.PLAIN_MESSAGE);
    return -1;  //vyskoci z metody, je li chyba
    }

try {
    BufferedReader vstup = new BufferedReader(new FileReader(vstSoubor));
    radek=vstup.readLine();
    vstup.close();
    }   catch (IOException v)
              {
              to.setText(to.getText()+v+'\n');
              }

for (int i=0; i<delka;i++)   //prevod na integer
    {
    if  (radek.charAt(i) == '0')        {vstData[i]=0;}
    else if (radek.charAt(i) == '1')    {vstData[i]=1; }
    else  {
          JOptionPane.showMessageDialog(null,"Vstupní data jsou v
nesprávném formátu!", "Chyba",JOptionPane.PLAIN_MESSAGE);
          return -1;
          }
    }
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// BLOK DEKODOVANI A URCENI SYNDROMU

int l=0, pocetDetChyb=0, pom=1;
int Jx[] = new int[3825];
int zbytek[] = new int[3825];

while (l<(delka))  //nacteni dat
      {
      Arrays.fill(Jx, 0);
      Arrays.fill(zbytek, 0);
      for (int k = 999;k >=0 ;k--)
          {
          Jx[k]=vstData[l];
          zbytek[k]=vstData[l];
          l++;
          }

      int poslJednicka = 999; //pred kazdym delenim nastavit na MSB


      while (poslJednicka > (23)) //vlastni deleni
            {
            int k2 = poslJednicka;
            for (int m = (delkaZabezpeceni); m >= 0; m--)
                  {
                  zbytek[k2] = (zbytek[k2] ^ Gx[m]);
                  k2--;
                  }
            poslJednicka = 0;

            for (int k3 = 0;k3 <  indexDelkaKodSlova;k3++)
                {
                if (zbytek[k3]==1) {  poslJednicka=k3;    }
                }


            }
      int syndrom = 0;

      for (int k4 = 0; k4 <delkaZabezpeceni; k4++)
          {
          syndrom += zbytek[k4];
          }



      if (syndrom != 0)     {pocetDetChyb++;
      }

      pom++;
      }
      return pocetDetChyb;


}
}
```